Policy-driven Middleware for Heterogeneous, Hybrid Cloud Platforms

Tom Desair, Wouter Joosen, Bert Lagaisse, Ansar Rafique, Stefan Walraven
iMinds-DistriNet, KU Leuven
3001 Leuven, Belgium
{firstname.lastname}@cs.kuleuven.be

ABSTRACT

The cloud computing paradigm promises increased flexibility and scalability. However, in private cloud environments this flexibility and scalability is constrained by the limited capacity. On the other hand, organizations are reluctant to migrate to public clouds because they lose control over their applications and data. The concept of a hybrid cloud tries to combine the benefits of private and public clouds, while also decreasing vendor lock-in.

This paper presents PaaSHopper, a middleware platform for hybrid cloud applications that enables organizations to keep fine-grained control over the execution of their applications. Driven by policies, the middleware dynamically decides which requests and tasks are executed in a particular part of the hybrid cloud. We validated this work by means of a prototype on top of a hybrid cloud consisting of a local JBoss AS cluster, Google App Engine, and Red Hat OpenShift.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Applications*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Design

Keywords

Hybrid cloud, Platform as a Service, Dynamic and context-aware adaptation

1. INTRODUCTION

Cloud computing is an emerging trend in the information technology industry and promises to deliver *"computing as a utility"*. This offers high flexibility and enables companies to quickly respond to changes in their markets with a limited (financial) risk.

There are three different service models within the cloud computing paradigm [8] referred to as Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). Each delivers a specific computational resource which is respectively virtualized infrastructure, a middleware platform or an application. Well-known examples for each category are Amazon EC2 [1] (IaaS), Google App Engine [5] (PaaS) and the Salesforce CRM application [15] (SaaS). In this paper we focus on Platform as a Service, or middleware as a service. PaaS aims to facilitate the development, deployment and hosting of cloud enterprise applications, while also offering the benefits of high availability and elastic scalability. Application providers do not have to take care of the infrastructure-related concerns, and therefore PaaS can play a major role for software vendors in the adoption of the cloud computing paradigm.

However, when using a public PaaS offering, companies lose partially control over their applications and data with the possibility of vendor lock-in. This is one of the main issues that withholds general cloud computing adoption [6, 11, 14]. Companies prefer their own private data center or private cloud, but this requires a big investment and offers less elasticity. The ideal solution would be a hybrid cloud setup that consists of the company's own private cloud and one or more public clouds. When the load on the private cloud becomes too high, several tasks can be delegated to the public clouds to alleviate the load. Moreover, hybrid cloud applications do not depend on the availability of a single cloud, which is a non-negligible risk [18, 4, 10]. Despite the fact that the interest in hybrid clouds greatly increases [7], the support for hybrid cloud applications is still fairly limited.

Concretely, developing hybrid cloud applications is complicated due to several challenges:

1. Heterogeneity and portability issues at development time,
2. Complex and (re)configurable decisions that need to be supported:
   (a) to enable smart and fine-grained deployment of application components across cloud environments,
   (b) to enable flexible execution locations at run time of certain tasks.

In essence, the different PaaS platforms actually all offer a lot of similar architectural concepts towards application developers, such as REST-based services, background workers and scalable storage. Often these storage services are based on NoSQL principles, for example scalable key-value

datastores and large-scale column-oriented storage. However, even the Java-based cloud platforms are very heterogeneous in terms of development API and deployment [17]. There is also a large difference in the trust relation between the customer and the different public cloud providers, based on different parameters: the location of the cloud provider, the business model of the cloud platform, or if the PaaS platform is the provider's core business or not.

In summary, there is a need for abstraction of the heterogeneity in development and deployment, as well as flexible and reconfigurable deployment and execution locations that are driven by policies. There is a need to specify constraints about which parts of the online application can run where (i.e. which part of the hybrid cloud), based on the properties of the context (e.g. customer and application), as well as properties of the cloud provider. The goal of this paper is to increase portability of the application, reduce the development overhead due to the heterogeneity, and also facilitate simultaneous, smart and scalable deployment and execution of the application components across multiple cloud providers.

In this paper, we present PaaSHopper, a policy-driven middleware that facilitates flexible development and scalable, adaptive execution of hybrid cloud applications on top of heterogeneous, Java-based PaaS platforms. The middleware offers an abstraction layer for common architectural concepts in PaaS platforms, such as background workers and large-scale blob storage, and also provides policy-driven fine-grained deployment and execution of the different component instances in the hybrid cloud application. Most related work, such as SCA [9] has focused on the heterogenity and portability problem. Therefore, for brevity, and to stay in the scope of this workshop, we focus on the policy-driven adaptive deployment and execution at runtime. The policies support constraints about the location of the datacenter, connection security, storage encryption, access to the cloud (public or private), workload, as well as other stakeholder-specific constraints.

The remainder of this paper is structured as follows. Section 2 describes the architecture of the middleware with respect to portability, interoperability and policy-driven component deployment. Section 3 presents a case study and an illustration with the prototype implementation. Section 4 compares our approach with related work and existing systems. Finally, Section 5 concludes this paper and indicates future research directions.

2. POLICY-DRIVEN MIDDLEWARE FOR HYBRID CLOUD PLATFORMS

In this section, we present a policy-driven middleware solution that enables organizations to keep control over the execution of their applications across hybrid clouds. The platform also offers built-in support for creating multi-tenant applications. A tenant is an organizational customer of the provided application, and a group of customers or employees of that organization are the end users of the provided application. A multi-tenant application will serve multiple tenants, and all end users of those tenants, on a shared application instance or shared pool of instances [3, 16]. The data of the different tenants and their specific configurations and policies are stored per tenant in a multi-tenant datastore.

First, we give an overview of the overall middleware architecture. Second, we focus on the abstraction layer and then on the policy-driven deployment layer.

2.1 Overview

An overview of our middleware solution to execute applications on hybrid cloud platforms using policies is presented in Fig. 1. Our middleware framework consists of an abstraction layer and a policy-driven distributed execution layer.

The abstraction layer constitutes a *portability driver component*, a *core component*, and an *interoperability component*. This layer tackles heterogeneity and interoperability requirements. The policy-driven deployment layer offers control over the execution of tasks by using policies and enables the control for each stakeholder. At this moment we support two types of policies: (i) *tenant policies* to define all tenant-specific constraints, and (ii) *monitoring policies* (by the application provider) to decide where the task will be executed based on load information. For example, the middleware supports maximum on-premise utilization when the load is low (to limit operational costs), and forces to also utilize the public cloud when the load reaches to a configured threshold.

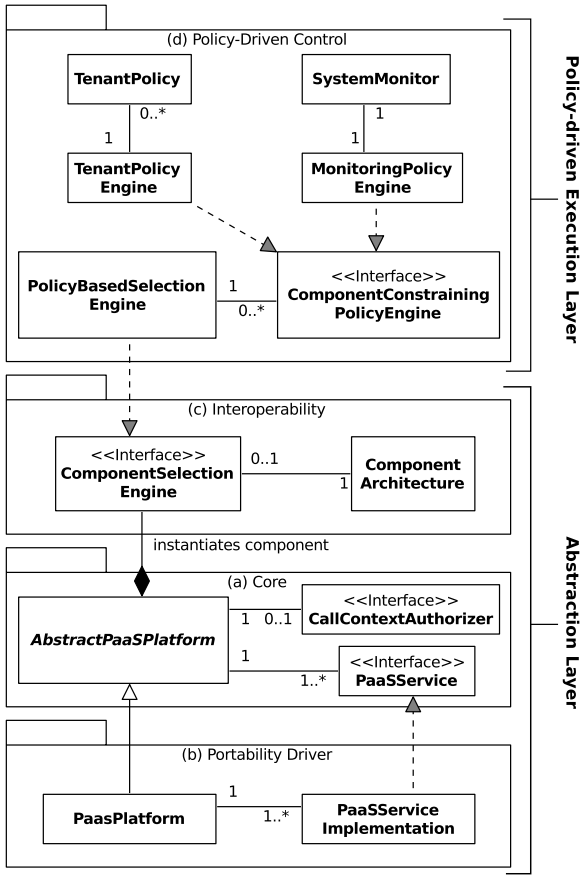

Figure 1: Overview of the PaaSHopper middleware with (a) a portability driver component, (b) the core component, (c) the interoperability component and (d) the policy-driven control component

2.2 Abstraction Layer

The abstraction layer is presented in Fig. 1 and consists of three components: (a) The *core component* provides a

uniform API to the application components for interaction to the middleware. (b) The *portability driver component* tackles heterogeneity problems and provides an implementation for the API of the core component for each of the different PaaS platforms. (c) The *interoperability component* is responsible for the transparent interoperability between the application components on different PaaS platforms.

The core concept is the AbstractPaaSPlatform which offers abstractions for the common architectural concepts and middleware services in PaaS platforms: structured data storage, blob data storage, http-based services and background workers executing asynchronous tasks. For brevity, we have omitted their details in this paper as these abstractions mainly solve the challenge of development-time heterogeneity and portability. For the rest of the paper we will focus on the run-time interoperability and policy-driven adaptive execution location for application components and tasks.

An application component interacts with the core component `AbstractPaaSPlatform` to retrieve references to other component instances. The core component uses a `ComponentSelectionEngine` instance of the interoperability component to select the appropriate component for the application. To be able to select an instance of a particular component type, our middleware uses an *architecture description*, which is loaded with the hybrid cloud application. This configuration file specifies the deployment of all the available local and remote instances for each component interface, including whether an instance is deployed on a public or private cloud. The application provider can also define certain properties and corresponding values for each instance. These properties reflect the characteristics of the underlying cloud platform. For example, if a certain component instance is deployed on a secured private cloud that uses encryption, these properties can be indicated in the configuration file as "secure" and "encrypted", both with value `true`. This is illustrated further in the next section.

After an instance is selected, it is returned to the application that can start invoking operations on it. The returned instance is either a local instance or a proxy to a remote instance, depending upon the policies and provided `CallContext`, as shown in Fig. 2. The `CallContext` also handles multi-tenancy (by means of a tenant ID) and contains all relevant information about the component call of a user of a certain tenant as shown in Fig. 3. The option of passing `CallContext` is imposed by the distributed nature of our middleware because in a hybrid cloud context there is no single application run-time environment or single main memory where information about the current active tenant can be stored. Explicitly passing `CallContext` simplifies the design and implementation and enables us to keep the different services stateless. The `CallContext` class also supports the implementation of an application-specific authorization mechanism (see Fig. 1): the application has to implement the `CallContextAuthorizer` interface (e.g. as part of a portability driver implementation). The abstract PaaS platform will then use the provided authorization service to authorize tentative calls based on the `CallContext` before retrieving a component instance.

2.3 Policy-driven Execution Layer

Our middleware offers control over the execution of a tenant's task. Based on the different tenant- and provider-specific policies it decides where in the hybrid cloud the task will be executed. Concretely, the hybrid application continuously needs to adapt the execution flow, for example depending on which tenant is executing. In our current middleware architecture (see component (d) in Fig. 1), policies are activated when the `PolicyBasedSelectionEngine` returns an instance of a requested component. The returned instance is either a local instance or a proxy to a remote instance. The `PolicyBasedSelectionEngine` manages a set of policies, and based on the constraints specified in these policies it returns a component that complies with these constraints. At this moment, we support two types of policies: tenant policies and monitoring policies.

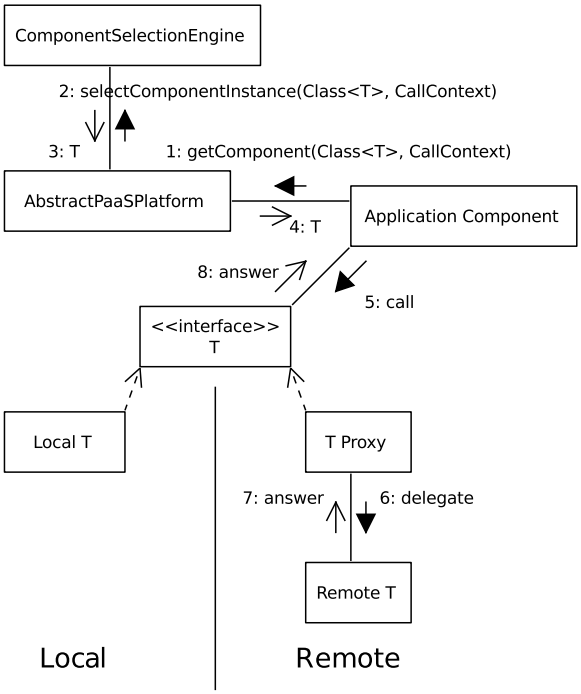

Figure 2: Transparent component retrieval

A *tenant policy* defines all tenant-specific constraints. These policies are automatically isolated from each other due to our multi-tenant data store abstraction. A tenant policy is coupled to the type of the message (e.g. confidential messages) that is sent to the requested component (and which is indicated by the `messageType` field in the `CallContext`, as illustrated in Fig. 3). For each message type, the tenant policy can list several requirements to which the receiving component must comply.

The second policy is the *monitoring policy* and is specified by the application provider. The monitoring policy engine dynamically decides where the task needs to be executed, either on-premise or remotely, based on the load information retrieved from the system monitor. When the load is low, the policy engine maximizes on-premise utilization and executes all tasks locally. When the load surpasses a configurable threshold (as defined in the monitoring policy), the policy engine tries to force a remote execution.

The specification of a policy is a set of constraints. Fig. 4 defines the grammar for describing tenant policies and their resulting constraints. The `PolicyBasedSelectionEngine` filters the set of all possible component instances that are matched by the different policies. The first component in-

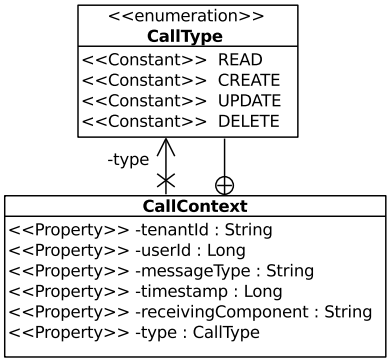

Figure 3: `CallContext` interface

stance that meets all imposed requirements is returned to core component of the hybrid cloud platform (cf. Fig. 2). Each policy gets assigned a unique priority integer value. This priority value is used when two policies output conflicting constraints. For example, when a tenant policy requires local execution but the monitoring policy forces remote execution due to the high load. In that case the policy with the highest priority overrules the other policy. In the middleware this ensures that only tenant tasks that are allowed to execute on a public cloud, are delegated when the load is high.

$$
\begin{array}{rcl}
\langle policy \rangle & ::= & \langle componentinfo \rangle\ , \\
 & & \langle messageinfo \rangle\ , \\
 & & \langle constraints \rangle \\
\langle componentinfo \rangle & ::= & \texttt{Component} = \langle interfacename \rangle \\
\langle messageinfo \rangle & ::= & \texttt{MessageType} = \langle value \rangle \\
 & | & \texttt{MessageType} = \texttt{Not}\ \langle value \rangle \\
\langle constraints \rangle & ::= & \langle locationinfo \rangle\ , \\
 & & \langle accessinfo \rangle\ , \\
 & & \langle properties \rangle \\
\langle locationinfo \rangle & ::= & \texttt{Location} = \langle location \rangle \\
\langle location \rangle & ::= & \texttt{Local} \mid \texttt{Remote} \\
 & | & \texttt{Unimportant} \\
\langle accessinfo \rangle & ::= & \texttt{Access} = \langle access \rangle \\
\langle access \rangle & ::= & \texttt{Public} \mid \texttt{Private} \\
 & | & \texttt{Unimportant} \\
\langle properties \rangle & ::= & \langle providerproperty \rangle = \langle value \rangle \\
 & | & \langle properties \rangle\ \texttt{and} \\
 & & \langle providerproperty \rangle = \langle value \rangle
\end{array}
$$

Figure 4: Tenant policy description grammar in BNF notation

Application providers can also implement their own control mechanisms. The `CallContext` contains much relevant information that enables other control mechanisms to be implemented (see Fig. 3). For example, an application can implement a user policy that allows to define individual policies on per end user (of tenants) basis. Furthermore, a custom implementation of the `ComponentSelectionEngine` interface can be provided and inserted into the abstract PaaS platform to support custom component selection.

When developing an application that implements multi-tenancy at the application level (the application itself is shared among tenants [16]), it is important to make the application components stateless because of the concurrent use by different tenants. Therefore it is necessary to enable stakeholder control on a per call basis. This implies that instances cannot be stored within the member variables of a class. While experimenting with the use of (traditional) dependency injection to inject the selected component in the client component as a member variable, it became clear to us that when executing a tenant call this always led to concurrency control issues. Dependencies thus need to be re-resolved for each call, as different policies apply for different tenants. For example, in [16] we also required a tenant-aware dependency injector to enable tenant-specific customization of SaaS applications.

3. ILLUSTRATION: THE CLOUDPOST CASE STUDY

We illustrate the proposed middleware architecture by means of a realistic case study: the CloudPost application. The CloudPost case study is a multi-tenant SaaS application for B2B document generation and provides online services to its tenants to generate, send, archive, and search large sets of customized digital documents. For example, such documents can be a large set of monthly bills from a telecom provider, or a mass commercial mailing from a retailer.

First, a tenant has to define one or more document templates that define the structure of a document. After the template definition, a tenant can upload raw data in an XML format and select a template. The raw data contain the customized content for a large set of documents of a certain type (e.g. bills) and for each document it contains metadata about the receiver and the delivery method. For example, the metadata can specifiy whether a document should be delivered by email or by normal mail. The CloudPost application will generate the documents by filling the chosen template with the raw data and then send them to the receiver by using the indicated delivery method. The application also archives each generated document so it makes it easy for a user to search them later.

Possible tenants of the CloudPost system can be *supermarkets* that send customized advertisements to their customers, *utility companies* that send out personal invoices, *hospitals* that want to deliver the medical reports to the doctors or patients, etc. Although each tenant has the same functional requirement, they have very different non-functional requirements with regards to security. All these tenants also have strict deadlines for sending out the documents, and all utility companies often send their bills out at the same moment (the so called end-of-month bill run). The datacenter with the on-premise services of CloudPost thus faces high peaks in load for short moments in time.

The problems of the CloudPost application are typically solved by using a hybrid cloud, where the public cloud is used as a spill-over for peaks in load from the private cloud. The PaaSHopper middleware allows the CloudPost provider to specify the properties of the different application component instances in the hybrid cloud (by means of the architecture description). For example, the document generation service in the private cloud provides secure communication, the service to create documents in one public cloud offers both encrypted storage and secure communication and the same service in another public cloud supports none of these properties (see Fig. 5).

```
<?xml version="1.0" encoding="UTF-8"?>
<components>
 <component id="Doc1">
  <interface>cloudpost.service.
      DocumentService</interface>
  <description>Generation and storage of
      documents</description>
  <implementation>
  cloudpost.document.generation.
      DocumentServiceImpl
  </implementation>
  <hosted>private</hosted>
  <properties>
   <secure>true</secure>
   <encrypted>false</encrypted>
  </properties>
 </component>
 <component id="Doc2">
  <interface>cloudpost.service.
      DocumentService</interface>
  <description>
  Generation and storage of documents on
      Google AppEngine
  </description>
  <implementation>remote</implementation>
  <url>http://cloudpost-gae.appspot.com/
      remote/docservice
  </url>
  <hosted>public</hosted>
  <properties>
   <secure>false</secure>
   <encrypted>false</encyrpted>
  </properties>
 </component>
 <component id="Doc3">
 <interface>cloudpost.service.
     DocumentService</interface>
  <description>
   Generation and storage of documents on
       OpenShift
  </description>
  <implementation>remote</implementation>
  <url>
   http://cloudpost.openshift.com/
       docservice
  </url>
  <hosted>public</hosted>
  <properties>
   <secure>true</secure>
   <encrypted>true</encyrpted>
  </properties>
 </component>
</components>
```

Figure 5: Example of a deployment descriptor for one private and two public versions of a component.

The tenant can define in a tenant policy which properties a certain application component must have. For example, for a confidential document type, the document generation service must provide encrypted storage or must run in a private cloud. An example of such a policy is shown in Fig. 6. In case of high workload, the PaaSHopper middleware will then generate non-confidential documents (like publicity) in the public cloud, and confidential documents in public clouds with encrypted storage, or in the private cloud.

The tenants can further define multiple document types (that are mapped to message types in the middleware) and they can define constraints for each type. For example, some

```
Component = cloudpost.service.
    DocumentService,
MessageType = confidential,
    Location = Unimportant,
    Access = Private,
        Encrypted = true
        and
        Secure = true
```

Figure 6: Example of a policy for confidential documents. Confidential documents should be processed in private clouds, or on an external location that supports encrypted storage and ensures secure communication.

banks even require that a private cloud is used in the same country as where the bank is located. This keeps the tenants in control of their documents, but also enables the CloudPost application to benefit from the elasticity of public clouds.

We have developed a Java prototype of the PaaSHopper middleware and an implementation of the CloudPost case study. We were able to deploy the CloudPost implementation on a hybrid cloud using the PaaSHopper middleware. The hybrid cloud consists of (i) local JBoss 7 Enterprise Server with the MongoDB for the storage, (ii) the Google App Engine [5] PaaS platform, (iii) the OpenShift [13] PaaS platform (on a Tomcat 7 gear extended with a MongoDB gear). Within the CloudPost implementation tenants are able to define document types and associated policies that restrict the set of locations where a document may be generated and stored.

4. RELATED WORK

Hybrid cloud applications. Paraiso et al. [11] present a *"federated multi-cloud PaaS infrastructure"* that makes it possible to deploy SCA applications on multiple heterogeneous PaaS and IaaS clouds. Their federated PaaS infrastructure relies on their own FraSCAti application environment for SCA applications. The infrastructure allows dynamic reconfiguration of component bindings and addition of components and services, but this happens globally for all the tenants. Their research focus is on both IaaS and PaaS clouds. However, for the IaaS clouds they deploy a preconfigured Tomcat application server, which is an approach we could also apply. Another key difference is that our research focuses on a technology-agnostic approach, without the need of an SCA runtime. In addition, we also support data multi-tenancy and the use of services of the underlying PaaS clouds whereas they do not consider these explicitly. The component binding reconfiguration mechanism of our middleware is activated on a per-tenant basis and is driven by tenant-specific policies and monitoring policies.

The research done by the European mOSAIC project is also related to ours [12]. The acronym mOSAIC stands for *"Open Source API and Platform for Multiple Clouds"*. This project develops a new PaaS platform with an open, independent API that offers support for heterogeneous hybrid clouds. Their PaaS platform can be deployed on top of various existing IaaS and PaaS clouds, which enables the portability of component-based applications developed for their platform. The components of deployed applications have to communicate via asynchronous messages. Their PaaS platform also uses a driver-based architecture in which they translate oper-

ations on their platform to the underlying platform by using multiple abstraction layers and a semantic ontology. The mOSAIC PaaS platform also automatically provisions cloud resources for deployed applications. While the researchers of the mOSAIC project focus on the creation of a new PaaS platform that automates a lot of tasks for its users (like cloud resource provisioning and scaling), our approach is creating thin and concise abstractions of the underlying platform. Our focus is on supporting more control for all application stakeholders by means of policies.

Policy-driven cloud middleware. If we look at the policy-driven aspect of our middleware, the research work by Azeez et al. [2] is also related. The authors design a middleware that supports the development of multi-tenant SOA applications. In such applications both tenants and application providers can define policies. They implement the multi-tenancy by making the different data services tenant-aware. Before executing a request, a data service fetches the current tenant information from the run-time environment context. The middleware uses a message dispatch mechanism in order to guide incoming request messages to the right service instance as indicated by global and tenant policies. The relevant policies are processed in two steps. In the first step all the global policies are applied and in the second step tenant-specific policies are taken into account. Besides the PaaS portability and hybrid cloud interoperability aspects, our research focus is more towards using policies in a hybrid cloud context. In such a context, there is no single run-time environment which contains information about the active tenant. Our policies focus on the choice of a PaaS platform within the hybrid cloud. Therefore, the PaaSHopper middleware supports more fine-grained policies that relate to the active tenant as well as the current call message type.

5. CONCLUSION

The use of hybrid cloud applications can solve many problems that enterprises currently have with cloud computing. However support for hybrid cloud applications is very limited. This paper proposes an architecture for a middleware that offers enhanced support for hybrid cloud applications by offering transparent cloud interoperability as well as increased control for all application stakeholders. The middleware is aimed at PaaS platforms, but also indirectly supports IaaS clouds through the use of application servers.

Many research opportunities are left open. In future work, we plan to support more PaaS platforms and perform a validation with other types of SaaS applications. Further, more adaptation support is required to dynamically (un)deploy components on the appropriate public platforms. Enabling more expressive policies is also an item for future research.

6. ACKNOWLEDGMENTS

This research is partially funded by the Research Fund KU Leuven, the iMinds DMS2 project and by the FWO project AIRCO. The iMinds DMS2 is a project co-funded by iMinds (Interdisciplinary institute for Technology), a research institute founded by the Flemish Government. Companies and organizations involved in the project are Agfa Healthcare, Luciad, UP-nxt and Verizon Terremark, with project support of IWT (government agency for Innovation by Science and Technology).

7. REFERENCES

[1] Amazon Web Services LLC. Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`. [Last visited on 30 August 2013].

[2] A. Azeez, S. Perera, D. Gamage, R. Linton, P. Siriwardana, D. Leelaratne, S. Weerawarana, and P. Fremantle. Multi-tenant SOA Middleware for Cloud Computing. In *IEEE International Conference on Cloud Computing*, pages 458–465, 2010.

[3] F. Chong and G. Carraro. Architecture Strategies for Catching the Long Tail. Microsoft Corporation, `http://msdn.microsoft.com/en-us/library/aa479069.aspx`, April 2006.

[4] D. Takahashi. Amazon's outage in third day: debate over cloud computing's future begins. VentureBeat. 2011.

[5] Google, Inc. Google App Engine. `http://code.google.com/appengine/`. [Last visited on 30 August 2013].

[6] N. Leavitt. Is Cloud Computing Really Ready for Prime Time? *Computer*, 42(1):15–20, 2009.

[7] N. Leavitt. Hybrid Clouds Move to the Forefront. *Computer*, 46(5):15–18, 2013.

[8] P. Mell and T. Grance. The NIST Definition of Cloud Computing. Special Publication 800-145, National Institute of Standards and Technology (NIST), September 2011.

[9] R. Mietzner, F. Leymann, and M. P. Papazoglou. Defining Composite Configurable SaaS Application Packages Using SCA, Variability Descriptors and Multi-tenancy Patterns. In *ICIW '08: 3rd International Conference on Internet and Web Applications and Services*, pages 156–161, June 2008.

[10] N. Bilton. Amazon web services knocked offline by storms. The New York Times (Bits). 2012.

[11] F. Paraiso, N. Haderer, P. Merle, R. Rouvoy, and L. Seinturier. A Federated Multi-cloud PaaS Infrastructure. In *IEEE International Conference on Cloud Computing*, pages 392–399, June 2012.

[12] D. Petcu, G. Macariu, S. Panica, and C. Crăciun. Portable Cloud applications – From theory to practice. *Future Generation Computer Systems*, 29(6):1417–1430, 2013.

[13] Red Hat, Inc. Red Hat OpenShift. `https://openshift.redhat.com/`. [Last visited on 30 August 2013].

[14] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi. Cloud computing – The business perspective. *Decis. Support Syst.*, 51(1):176–189, 2011.

[15] Salesforce.com, Inc. Salesforce CRM. `http://www.salesforce.com/`. [Last visited on 30 August 2013].

[16] S. Walraven, E. Truyen, and W. Joosen. A middleware layer for flexible and cost-efficient multi-tenant applications. In *Middleware '11: Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware*, pages 370–389. 2011.

[17] S. Walraven, E. Truyen, and W. Joosen. Comparing PaaS offerings in light of SaaS development. *Computing*, pages 1–56, 2013.

[18] Z. Whittaker. Amazon cloud down; reddit, github, other major sites affected. ZDNet.com. 2012.